北京理工大學
BEIJING INSTITUTE OF TECHNOLOGY

大数据泛构实验报告 2

学　　院： 珠海校区

专　　业： 计算机科学与技术

学　　号： 3120256739

姓　　名： 丁纪翔

任课教师： 毛睿

2025 年 11 月 16 日

目录

1. 引言 4

1.1 研究背景与意义 4

1.2 任务回顾与目标 4

1.2.1 Assignment 1 工作简述 4

1.2.2 Assignment 2 核心目标 5

1.3 实验环境 5

1.4 报告结构 6

2. 相似性查询与索引系统设计 6

2.1 系统架构扩展 6

2.1.1 在原有架构中集成查询与索引模块 6

2.1.2 更新后的系统模块划分 7

2.2 线性扫描查询算法设计 7

2.2.1 范围查询 (Range Query) 算法原理 7

2.2.2 k 近邻查询 (kNN) 算法原理 9

2.2.3 多样化 k 近邻查询 (dkNN) 算法原理 10

2.3 Pivot Table 索引设计 11

2.3.1 核心思想：基于三角不等式的剪枝 11

2.3.2 数据结构设计与构建流程 12

2.3.3 基于 Pivot Table 的范围查询流程 13

2.3.4 基于 Pivot Table 的 kNN 查询流程 13

3. 核心功能实现 14

3.1 线性扫描查询实现 14

3.1.1 范围查询 (Range Query) 核心代码 14

3.1.2 k 近邻查询 (kNN) 核心代码 16

3.2 Pivot Table 索引实现 17

3.2.1 Pivot Table 构建核心代码 17

3.2.2 支撑点选择 (FFT 策略) 实现 18
3.2.3 基于 Pivot Table 的范围查询实现 20

4. 功能正确性验证 22
4.1 测试环境与数据集 22
4.2 线性扫描查询正确性验证 23
4.2.1 范围查询 (Range Query) 测试 23
4.2.2 k 近邻查询 (kNN) 测试 24
4.2.3 多样化 k 近邻查询 (dkNN) 测试 24
4.3 Pivot Table 索引正确性验证 25
4.3.1 构建过程验证 25
4.3.2 基于索引的查询结果与线性扫描结果对比 25

5. 性能分析与探索 26
5.1 性能评估指标定义 26
5.2 实验设计 27
5.2.1 数据集与查询集选择 27
5.2.2 支撑点选择策略 27
5.3 支撑点数量对性能的影响分析 28
5.3.1 范围查询性能 28
5.3.2 kNN 查询性能 29
5.4 支撑点选择策略对性能的影响分析 30
5.5 查询性能对比：线性扫描 vs. Pivot Table 索引 31
5.5.1 范围查询性能对比 31
5.5.2 kNN 查询性能对比 32
5.6 性能分析总结 32

6. 总结与展望 33
6.1 工作总结 33
6.1.1 核心功能实现 33

6.1.2 正确性验证 34
6.1.3 性能分析成果 34
6.1.4 理论理解深化 34
6.2 系统不足与改进方向 35
6.2.1 性能优化 35
6.2.2 功能扩展 36
6.2.3 实验深化 36
6.2.4 工程实践 37
6.3 展望 37

1. 引言

1.1 研究背景与意义

在大数据时代，数据的多样性 (Variety) 是大数据三大特征之一。面对图片、文本、音频、生物序列等多种数据类型，传统的专用数据处理系统需要为每种数据类型量身定做，开发和维护成本巨大。度量空间 (Metric Space) 作为一种统一的数据抽象，为构建通用的数据管理与分析系统提供了理论基础。

度量空间的核心思想是：只要数据之间可以定义满足度量空间性质 (非负性、对称性、三角不等性) 的距离函数，就可以使用相同的算法处理不同类型的数据。这种”求同存异” 的思想，使得我们可以用统一的相似性查询和索引技术处理多种数据类型，大大降低了系统的开发和维护成本。

相似性查询是度量空间数据管理的核心功能，广泛应用于图像检索、推荐系统、异常检测等场景。然而，随着数据规模的增长，线性扫描的查询方式效率低下。索引技术通过预计算和剪枝策略，可以显著减少距离计算次数，提升查询性能。Pivot Table(支撑点表) 是一种经典的度量空间索引结构，利用三角不等式进行剪枝，在保证查询准确性的同时大幅提升效率。

1.2 任务回顾与目标

1.2.1 Assignment 1 工作简述

在 Assignment 1 中，我们构建了度量空间数据管理的基础框架：

1. **核心抽象类设计**：设计了 `MetricSpaceData` 抽象父类和 `MetricFunction` 接口，为不同数据类型提供统一的抽象层。

2. **向量数据类型**：实现了 `VectorData` 类，支持任意维度的向量数据，以及基于闵可夫斯基距离 (Minkowski Distance) 的距离计算，包括 L_1(曼哈顿距离)、L_2(欧几里得距离) 和 L_∞(切比雪夫距离)。

3. **蛋白质序列类型**：实现了 `ProteinData` 类，支持蛋白质序列的 6-mers 片段表示，以及基于 mPAM250a 替代矩阵的全局序列比对距离 (Alignment Distance)。

4. **数据读取功能**：实现了向量数据和蛋白质序列数据的读取器，支持 UMAD 数据集格式。

1.2.2 Assignment 2 核心目标

在 Assignment 1 的基础上，Assignment 2 聚焦于度量空间的相似性查询与索引：

1. **线性扫描查询**：实现三种基本的相似性查询算法——范围查询 (Range Query)、k 近邻查询 (kNN Query) 和多样化 k 近邻查询 (dkNN Query)。
2. **Pivot Table 索引**：实现基于支撑点表的索引结构，利用三角不等式进行剪枝优化，减少距离计算次数。
3. **功能正确性验证**：设计易于验证的测试用例，确保查询和索引功能的正确性。
4. **性能分析与探索**：深入分析支撑点数量和选择策略对查询性能的影响，揭示 Pivot Table 索引的加速效果。

1.3 实验环境

本实验的软硬件环境如下：

- **操作系统**：Windows 11 (10.0.26100)
- **开发语言**：Java 18.0.2.1
- **构建工具**：Apache Maven 3.9.11
- **开发环境**：VS Code
- **测试框架**：JUnit 4.13.2
- **数据集**：UMAD (Universal Management and Analysis of Data)
  - Uniform 20-d vector：20 维均匀分布向量，100 万个数据点
  - 测试规模：1000, 5000, 10000 个数据点
- **硬件配置**：Intel Core 处理器，8GB 内存

1.4 报告结构

本报告共分为 6 章：

- **第 1 章引言**：介绍研究背景、任务目标和实验环境。
- **第 2 章系统设计**：详细阐述线性扫描查询和 Pivot Table 索引的设计原理。
- **第 3 章核心功能实现**：展示关键代码实现和技术细节。
- **第 4 章功能正确性验证**：通过测试用例验证系统功能的正确性。
- **第 5 章性能分析与探索**：分析支撑点数量、选择策略对性能的影响。
- **第 6 章总结与展望**：总结工作成果，展望未来改进方向。

2. 相似性查询与索引系统设计

2.1 系统架构扩展

2.1.1 在原有架构中集成查询与索引模块

在 Assignment 1 构建的基础架构之上，我们扩展了两个核心模块：

1. **查询模块 (query 包)**：实现三种线性扫描查询算法
   - `LinearScanRangeQuery`：范围查询
   - `LinearScanKNNQuery`：k 近邻查询
   - `LinearScanDKNNQuery`：多样化 k 近邻查询
2. **索引模块 (index 包)**：实现 Pivot Table 索引结构
   - `PivotTable`：索引数据结构
   - `PivotSelector`：支撑点选择器
   - `PivotTableRangeQuery`：基于索引的范围查询

- `PivotTableKNNQuery`：基于索引的 kNN 查询

这两个模块与 Assignment 1 的核心抽象类 (`MetricSpaceData`、`MetricFunction`) 和数据类型实现 (`VectorData`、`ProteinData`) 无缝集成，体现了度量空间通用数据处理的设计思想。

2.1.2 更新后的系统模块划分

完整的系统架构分为五层：

1. **核心抽象层 (core 包)**：定义度量空间数据和距离函数的抽象接口
2. **数据类型层 (datatype 包)**：实现具体的数据类型和距离函数
3. **数据 IO 层 (io 包)**：负责数据集的读取和解析
4. **查询层 (query 包)**：实现线性扫描查询算法
5. **索引层 (index 包)**：实现 Pivot Table 索引和基于索引的查询

该架构遵循” 依赖倒置原则”，高层模块 (查询、索引) 依赖于抽象 (核心接口)，而非具体实现，保证了系统的可扩展性和通用性。

2.2 线性扫描查询算法设计

线性扫描 (Linear Scan) 是最直观的查询方法，通过遍历数据集中的每个对象，逐一计算与查询对象的距离，筛选出满足条件的结果。虽然效率较低，但线性扫描保证了结果的准确性，常用作验证索引查询正确性的基准。

2.2.1 范围查询 (Range Query) 算法原理

定义：给定查询对象 q 和查询半径 r，范围查询返回数据集 S 中所有与 q 距离不超过 r 的对象：

$$RQ(q, r) = \{s \in S \mid d(q, s) \leq r\}$$

其中 d 是度量空间中的距离函数。

算法流程：

1. 初始化结果集 R 为空集
2. 遍历数据集 S 中的每个对象 s
3. 计算查询对象 q 与 s 的距离 $d(q, s)$
4. 如果 $d(q, s) \leq r$，将 s 加入结果集 R
5. 返回结果集 R

复杂度分析：

- 时间复杂度：$O(n)$，其中 n 是数据集大小
- 距离计算次数：n 次
- 空间复杂度：$O(k)$，其中 k 是结果集大小

示例：假设数据集包含 5 个二维向量，使用 L_2 距离，查询对象为原点 $(0, 0)$，查询半径 $r = 1.5$：

数据对象	与原点距离	是否在结果中
$(0, 0)$	0	✓
$(1, 0)$	1	✓
$(0, 1)$	1	✓
$(3, 4)$	5	✗
$(5, 5)$	7.07	✗

表 2-1 范围查询示例

2.2.2 k 近邻查询 (kNN) 算法原理

定义：给定查询对象 q 和整数 k，kNN 查询返回数据集 S 中距离 q 最近的 k 个对象：

$$kNN(q,k) = \{k\text{个最近邻} \mid \forall s \in kNN(q,k), \forall s' \notin kNN(q,k) : d(q,s) \leq d(q,s')\}$$

算法流程：

1. 创建最大堆 H(优先队列)，用于维护当前 k 个最近邻
2. 遍历数据集 S 中的每个对象 s
3. 计算查询对象 q 与 s 的距离 $dist = d(q,s)$
4. 如果 $|H| < k$，直接将 $(s, dist)$ 插入堆 H
5. 否则，如果 $dist < H.top().distance$，移除堆顶，插入 $(s, dist)$
6. 将堆 H 转换为按距离升序排列的列表并返回

复杂度分析：

- 时间复杂度：$O(n \log k)$
- 距离计算次数：n 次
- 空间复杂度：$O(k)$

关键数据结构：使用最大堆维护当前 k 个最近邻，堆顶元素是当前 k 个邻居中距离最大的。这样每次只需与堆顶比较，如果新对象距离更小，则替换堆顶，保持堆中始终是距离最小的 k 个对象。

2.2.3 多样化 k 近邻查询 (dkNN) 算法原理

动机：传统 kNN 可能返回非常相似的对象 (如都来自同一聚类)，缺乏多样性。例如，查询" 苹果" 的图片，传统 kNN 可能返回 10 张红苹果，而 dkNN 会返回红苹果、青苹果、苹果树等多样化的结果。

定义：多样化 kNN 在保证相关性 (与查询对象距离小) 的同时，最大化结果集的多样性 (结果之间距离大)。定义得分函数：

$$score(c) = (1 - \lambda) \cdot (-d(q, c)) + \lambda \cdot \min_{r \in R} d(c, r)$$

其中：

- $\lambda \in [0, 1]$ 是多样性权重
- $d(q, c)$ 是候选对象 c 与查询对象 q 的距离 (相关性项)
- $\min_{r \in R} d(c, r)$ 是 c 与当前结果集 R 中最近对象的距离 (多样性项)

算法流程 (贪心策略)：

1. 执行 kNN 查询，获取 $k \times \alpha$ 个候选 ($\alpha > 1$，如 $\alpha = 2$)
2. 初始化结果集 R 为空
3. 将距离查询对象最近的候选加入 R
4. 当 $|R| < k$ 且还有候选时：
    - 对每个候选 c，计算得分 $score(c)$
    - 选择得分最高的候选加入 R
5. 返回结果集 R

参数影响：

- $\lambda = 0$：退化为传统 kNN，只考虑相关性
- $\lambda = 1$：完全追求多样性，可能牺牲相关性
- $\lambda = 0.5$：平衡相关性和多样性

2.3 Pivot Table 索引设计

Pivot Table 是一种基于距离预计算和三角不等式剪枝的度量空间索引结构，其核心思想是通过少量的距离计算和大量的数值比较来加速查询。

2.3.1 核心思想：基于三角不等式的剪枝

三角不等式：度量空间中的距离函数满足：

$$d(x,z) \leq d(x,y) + d(y,z)$$

等价形式：

$$d(x,z) \geq |d(x,y) - d(y,z)|$$

剪枝原理：利用三角不等式，可以在不计算实际距离的情况下判断某些数据对象是否满足查询条件。

排除规则 (Exclusion Rule)：对于支撑点 p、查询对象 q、数据对象 s 和查询半径 r：

$$\text{如果} \quad |d(p,q) - d(p,s)| > r \quad \Rightarrow \quad d(q,s) > r$$

证明：由三角不等式 $d(q,s) \geq |d(p,q) - d(p,s)|$，若 $|d(p,q) - d(p,s)| > r$，则必有 $d(q,s) > r$，因此 s 不在查询结果中，可以直接排除。

包含规则 (Inclusion Rule)：

$$\text{如果} \quad d(p,q) + d(p,s) \leq r \quad \Rightarrow \quad d(q,s) \leq r$$

证明：由三角不等式 $d(q,s) \leq d(q,p)+d(p,s) = d(p,q)+d(p,s)$，若 $d(p,q)+d(p,s) \leq r$，则必有 $d(q,s) \leq r$，因此 s 一定在查询结果中，可以直接包含。

几何直观：以支撑点 p 为中心：

- 若 s 在以 p 为中心、半径为 $d(p,s)$ 的圆上
- 若 q 在以 p 为中心、半径为 $d(p,q)$ 的圆上

- 当 $|d(p,q) - d(p,s)|$ 很大时，s 必然在以 q 为中心、半径为 r 的查询圆之外

2.3.2 数据结构设计与构建流程

数据结构：

Pivot Table 包含三个核心组件：

```
public class PivotTable {
    private List<MetricSpaceData> pivots;      // k个支撑点
    private List<MetricSpaceData> data;        // n个数据对象
    private double[][] distanceTable;          // n×k距离表
    private MetricFunction metric;             // 距离函数

    // distanceTable[i][j] = d(data[i], pivots[j])
}
```

Listing 1: Pivot Table 数据结构

构建流程：

1. **支撑点选择：**根据选择策略 (RANDOM/FFT/CENTER/BORDER) 从数据集中选择 k 个支撑点
2. **距离预计算：**计算每个数据对象到每个支撑点的距离，填充距离表
3. **存储：**保存支撑点列表、数据列表和距离表

构建复杂度：

- 距离计算次数：$n \times k$ 次
- 存储空间：$O(n \times k)$
- 对于 FFT 选择策略，支撑点选择需要额外 $O(n \times k^2)$ 次距离计算

2.3.3 基于 Pivot Table 的范围查询流程

查询流程：

1. **预计算**：计算查询对象 q 到所有支撑点的距离 $dpq[j] = d(q, pivots[j])$，共 k 次距离计算
2. **剪枝判断**：对每个数据对象 $data[i]$：
   - 遍历每个支撑点 j，获取 $dps = distanceTable[i][j]$
   - **包含规则**：若 $dpq[j] + dps \leq r$，直接将 $data[i]$ 加入结果，跳过该对象
   - **排除规则**：若 $|dpq[j] - dps| > r$，标记该对象为剪枝，跳过该对象
3. **实际距离计算**：对无法通过剪枝判断的对象，计算 $d(q, data[i])$，若 $\leq r$ 则加入结果
4. **返回结果集**

性能优势：

- 理想情况：若所有对象都被剪枝，距离计算仅 k 次
- 实际情况：距离计算 $k + \alpha \times n$ 次，其中 $\alpha \in [0, 1]$ 是未被剪枝的比例
- 剪枝率：$(1 - \alpha) \times 100\%$，通常可达 70%-99%

2.3.4 基于 Pivot Table 的 kNN 查询流程

kNN 查询的关键改进是**动态查询半径**策略：

查询流程：

1. 初始化最大堆 H 和当前查询半径 $r_{current} = +\infty$
2. 预计算查询对象 q 到所有支撑点的距离 $dpq[j]$
3. 对每个数据对象 $data[i]$：

- 使用 $r_{current}$ 进行排除规则剪枝：若 $|dpq[j] - dps| > r_{current}$，跳过
- 否则计算实际距离 $dist = d(q, data[i])$
- 更新堆 H：
  - 若 $|H| < k$,插入 $(data[i], dist)$,当 $|H| = k$ 时更新 $r_{current} = H.top().distance$
  - 若 $dist < r_{current}$，移除堆顶，插入 $(data[i], dist)$，更新 $r_{current}$

4. 返回堆 H 中的 k 个最近邻

动态半径的优势：

- 随着查询进行，发现更近的邻居，$r_{current}$ 不断缩小
- $r_{current}$ 越小，排除规则的剪枝效果越好
- 后期遍历的对象被剪枝的概率更高

与范围查询的区别：

- 范围查询：半径 r 固定
- kNN 查询：半径 $r_{current}$ 动态变化，从 $+\infty$ 逐渐缩小到第 k 个最近邻的距离

3. 核心功能实现

本章展示系统的核心代码实现，重点说明关键算法和技术细节。由于完整代码较长，此处展示关键片段。

3.1 线性扫描查询实现

3.1.1 范围查询 (Range Query) 核心代码

范围查询的实现遍历数据集，计算每个对象与查询对象的距离，筛选出距离不超过半径的对象。

```java
public class LinearScanRangeQuery {
    public static List<MetricSpaceData> execute(
            List<? extends MetricSpaceData> dataset,
            RangeQuery query,
            MetricFunction metric) {

        List<MetricSpaceData> results = new ArrayList<>();
        MetricSpaceData queryObject = query.getQueryObject();
        double radius = query.getRadius();

        // 遍历数据集中的每个对象
        for (MetricSpaceData data : dataset) {
            // 计算距离
            double distance = metric.getDistance(queryObject,
                data);

            // 判断是否在查询半径内
            if (distance <= radius) {
                results.add(data);
            }
        }

        return results;
    }
}
```

Listing 2: 线性扫描范围查询实现

关键点：

- 使用泛型 `List<? extends MetricSpaceData>` 支持所有度量空间数据类型
- 通过 `MetricFunction` 接口调用距离函数，实现算法的通用性

- 时间复杂度 $O(n)$，距离计算次数为 n

3.1.2 k 近邻查询 (kNN) 核心代码

kNN 查询使用优先队列 (最大堆) 维护当前 k 个最近邻。

```
public class LinearScanKNNQuery {
    public static List<KNNResult> execute(
            List<? extends MetricSpaceData> dataset,
            KNNQuery query,
            MetricFunction metric) {

        MetricSpaceData queryObject = query.getQueryObject();
        int k = query.getK();

        // 最大堆:堆顶是当前k个邻居中距离最大的
        PriorityQueue<KNNResult> maxHeap = new PriorityQueue<>(
            (a, b) -> Double.compare(b.getDistance(), a.
                getDistance())
        );

        // 遍历数据集
        for (MetricSpaceData data : dataset) {
            double distance = metric.getDistance(queryObject,
                data);

            if (maxHeap.size() < k) {
                // 堆未满,直接加入
                maxHeap.offer(new KNNResult(data, distance));
            } else if (distance < maxHeap.peek().getDistance())
                {
                // 找到更近的邻居,替换堆顶
                maxHeap.poll();
                maxHeap.offer(new KNNResult(data, distance));
```

```
            }
        }

        // 转换为升序列表
        List<KNNResult> results = new ArrayList<>(maxHeap);
        Collections.sort(results,
            (a, b) -> Double.compare(a.getDistance(), b.
                getDistance()));

        return results;
    }
}
```

Listing 3: 线性扫描 kNN 查询实现

关键点：

- 使用最大堆保持堆中是距离最小的 k 个对象，堆顶是第 k 小的距离
- 每次只需与堆顶比较，避免对所有结果排序，时间复杂度 $O(n \log k)$
- `KNNResult` 封装了数据对象和距离，方便结果处理

3.2 Pivot Table 索引实现

3.2.1 Pivot Table 构建核心代码

```
public class PivotTable {
    private List<MetricSpaceData> pivots;
    private List<MetricSpaceData> data;
    private double[][] distanceTable;
    private MetricFunction metric;

    public PivotTable(List<? extends MetricSpaceData> dataset,
                      int pivotCount,
```

```
                        MetricFunction metric,
                        PivotSelectionMethod method) {
        this.data = new ArrayList<>(dataset);
        this.metric = metric;

        // 选择支撑点
        this.pivots = PivotSelector.selectPivots(
            dataset, pivotCount, metric, method);

        // 构建距离表
        buildDistanceTable();
    }

    private void buildDistanceTable() {
        int n = data.size();
        int k = pivots.size();
        distanceTable = new double[n][k];

        // 计算每个数据对象到每个支撑点的距离
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < k; j++) {
                distanceTable[i][j] =
                    metric.getDistance(data.get(i), pivots.get(
                        j));
            }
        }
    }
}
```

Listing 4: Pivot Table 构建实现

3.2.2 支撑点选择 (FFT 策略) 实现

```java
private static List<MetricSpaceData> farthestFirstTraversal(
        List<? extends MetricSpaceData> dataset,
        int pivotCount,
        MetricFunction metric) {

    List<MetricSpaceData> pivots = new ArrayList<>();

    // 第一个支撑点:随机选择
    pivots.add(dataset.get(new Random().nextInt(dataset.size())
       ));

    // 选择剩余k-1个支撑点
    for (int i = 1; i < pivotCount; i++) {
        MetricSpaceData farthest = null;
        double maxMinDist = -1;

        // 找到距离已选支撑点最远的点
        for (MetricSpaceData candidate : dataset) {
            if (pivots.contains(candidate)) continue;

            // 计算到已选支撑点的最小距离
            double minDist = Double.MAX_VALUE;
            for (MetricSpaceData pivot : pivots) {
                double dist = metric.getDistance(candidate,
                   pivot);
                minDist = Math.min(minDist, dist);
            }

            // 更新最远点
            if (minDist > maxMinDist) {
                maxMinDist = minDist;
                farthest = candidate;
```

```
            }
        }

        pivots.add(farthest);
    }

    return pivots;
}
```

Listing 5: FFT 支撑点选择算法

关键点：

- FFT 策略每次选择距离已选支撑点最远的点，保证支撑点分布均匀
- 构建复杂度：距离表 $O(nk)$，FFT 选择 $O(nk^2)$
- 距离表使用二维数组存储，空间 $O(nk)$

3.2.3 基于 Pivot Table 的范围查询实现

```
public class PivotTableRangeQuery {
    public static List<MetricSpaceData> execute(
            PivotTable pivotTable,
            RangeQuery query) {

        List<MetricSpaceData> results = new ArrayList<>();
        MetricSpaceData queryObject = query.getQueryObject();
        double radius = query.getRadius();

        List<MetricSpaceData> pivots = pivotTable.getPivots();
        List<MetricSpaceData> data = pivotTable.getData();
        double[][] distanceTable = pivotTable.getDistanceTable
            ();
        MetricFunction metric = pivotTable.getMetric();
```

```
14
15          // 预计算:查询对象到所有支撑点的距离
16          double[] dpq = new double[pivots.size()];
17          for (int j = 0; j < pivots.size(); j++) {
18              dpq[j] = metric.getDistance(queryObject, pivots.get
                    (j));
19          }
20
21          // 对每个数据对象进行剪枝判断
22          for (int i = 0; i < data.size(); i++) {
23              boolean pruned = false;
24              boolean included = false;
25
26              // 遍历所有支撑点
27              for (int j = 0; j < pivots.size(); j++) {
28                  double dps = distanceTable[i][j];
29
30                  // 包含规则
31                  if (dpq[j] + dps <= radius) {
32                      results.add(data.get(i));
33                      included = true;
34                      break;
35                  }
36
37                  // 排除规则
38                  if (Math.abs(dpq[j] - dps) > radius) {
39                      pruned = true;
40                      break;
41                  }
42              }
43
44              // 无法剪枝,计算实际距离
45              if (!pruned && !included) {
```

```
46                  double actualDist = metric.getDistance(
47                          queryObject, data.get(i));
48                  if (actualDist <= radius) {
49                      results.add(data.get(i));
50                  }
51              }
52          }
53
54          return results;
55      }
56  }
```

Listing 6: Pivot Table 范围查询

4. 功能正确性验证

本章通过设计易于验证的测试用例，验证线性扫描查询和 Pivot Table 索引的功能正确性。

4.1 测试环境与数据集

测试环境：

- JUnit 4.13.2 测试框架
- 数据集：Uniform 20-d vector
- 测试规模：10 个数据点 (小规模，便于人工验证)
- 距离函数：L_2 欧几里得距离

测试策略：

1. 使用小规模数据集，输出详细的计算过程
2. 对比线性扫描和 Pivot Table 索引的查询结果

3. 验证结果集大小、对象和距离的一致性

4. 验证度量空间三大性质 (非负性、对称性、三角不等性)

4.2 线性扫描查询正确性验证

4.2.1 范围查询 (Range Query) 测试

测试用例：数据集包含 10 个 20 维向量，查询对象为第一个向量，查询半径分别为 0.05、0.10、0.15、0.20。

测试输出 (部分)：

```
==================================================
线性扫描范围查询测试
==================================================

【测试配置】
  数据集大小: 10 个向量
  查询对象: Vector[0]
  距离函数: L2 (欧几里得距离)

【查询半径 r = 0.05】
  计算距离过程:
    d(q, data[0]) = 0.0000    <=  0.05  [包含]
    d(q, data[1]) = 0.8654    >   0.05  [排除]
    d(q, data[2]) = 0.7923    >   0.05  [排除]
    ...
  结果数量: 1
  距离计算次数: 10

【查询半径 r = 0.10】
```

```
结果数量: 1
距离计算次数: 10
```

验证结论：

- ✓ 所有测试用例通过
- ✓ 结果集大小随半径增大而增大或不变
- ✓ 距离计算次数恒为数据集大小 $n = 10$
- ✓ 查询对象自身始终在结果集中 (距离为 0)

4.2.2 k 近邻查询 (kNN) 测试

测试用例：数据集 10 个向量，k 分别为 5、10、20、50。

验证结论：

- ✓ 结果按距离升序排列
- ✓ $k \leq n$ 时返回 k 个结果，$k > n$ 时返回全部 n 个结果
- ✓ 第一个结果是查询对象自身 (距离为 0)
- ✓ 相邻结果的距离单调不减

4.2.3 多样化 k 近邻查询 (dkNN) 测试

测试用例：$k = 5$，多样性权重 $\lambda = 0, 0.5, 0.8$。

验证结论：

- ✓ $\lambda = 0$ 时结果与传统 kNN 完全一致
- ✓ 随 λ 增大，结果的平均成对距离增大，多样性提高
- ✓ 第一个结果始终是最近邻 (贪心策略起点)

4.3 Pivot Table 索引正确性验证

4.3.1 构建过程验证

验证结论：

- ✓ FFT 策略选择的支撑点之间距离较大，分布均匀
- ✓ 距离表大小为 $n \times k$
- ✓ 距离表元素满足度量空间性质 (非负性、对称性)

4.3.2 基于索引的查询结果与线性扫描结果对比

测试策略：对相同的查询参数，对比线性扫描和 Pivot Table 索引的查询结果。

范围查询对比：

```
【查询参数】
  查询对象: Vector[0]
  查询半径: 0.15
  支撑点数量: 5

【一致性检验】
  结果数量: 一致
  结果对象: 一致
```

kNN 查询对比：

```
【查询参数】
  查询对象: Vector[0]
  k: 5
  支撑点数量: 5

【一致性检验】
```

结果数量：一致
结果对象：一致
结果顺序：一致
距离值：一致

验证结论：

- ✓ 所有测试用例中，Pivot Table 索引结果与线性扫描结果完全一致
- ✓ 结果集大小、对象、顺序、距离值全部匹配
- ✓ Pivot Table 索引在保证结果正确性的同时，减少了距离计算次数
- ✓ 范围查询的剪枝率通常高于 kNN 查询

5. 性能分析与探索

本章通过系统的实验设计，深入分析支撑点数量、选择策略对 Pivot Table 查询性能的影响，并对比线性扫描与索引查询的性能差异。

5.1 性能评估指标定义

为了全面评估查询性能，我们定义以下关键指标：

1. **查询时间 (Query Time)**：执行一次查询所需的时间，单位毫秒 (ms)。
2. **平均距离计算次数 (Average Distance Calculations)**：执行查询时调用距离函数的次数，是主要计算开销的来源。
3. **剪枝数 (Pruned Count)**：通过三角不等式剪枝规则排除的数据对象数量。
4. **剪枝率 (Pruning Rate)**：

$$\text{剪枝率} = \frac{\text{剪枝数量}}{\text{数据集大小}} \times 100\%$$

剪枝率越高，说明索引效果越好。

5. **加速比 (Speedup)**：

$$\text{加速比} = \frac{\text{线性扫描时间}}{\text{索引查询时间}}$$

加速比大于 1 说明索引带来了性能提升。

6. **结果数量 (Result Count)**：查询返回的数据对象数量，用于验证查询正确性。

5.2 实验设计

5.2.1 数据集与查询集选择

数据集：

- 来源：UMAD Uniform 20-d vector 数据集
- 维度：20 维均匀分布向量
- 距离函数：L_2 欧几里得距离
- 测试规模：1000, 5000, 10000 个数据点

查询集：

- 查询对象：从数据集中随机选择
- 查询次数：每个配置执行 10 次查询，报告平均值
- 范围查询半径：0.05, 0.10, 0.15, 0.20
- kNN 查询的 k 值：5, 10, 20, 50

5.2.2 支撑点选择策略

本实验对比两种支撑点选择策略：

1. **RANDOM(随机选择)**：
    - 优点：构建速度快，无额外距离计算

- 缺点：支撑点可能聚集，剪枝效果不稳定

2. **FFT(最远优先遍历)：**

- 优点：支撑点分布均匀，覆盖整个数据空间，剪枝效果好
- 缺点：构建开销较大，需要 $O(nk^2)$ 次距离计算

5.3 支撑点数量对性能的影响分析

本实验固定数据集大小为 5000，支撑点选择策略为 FFT，测试不同支撑点数量 (5, 10, 15, 20, 25, 30) 对查询性能的影响。

5.3.1 范围查询性能

实验配置：

- 数据集大小：5000 个向量
- 支撑点数量：5, 10, 15, 20, 25, 30
- 查询半径：0.05, 0.10, 0.15, 0.20
- 查询次数：每个配置 10 次取平均

实验结果 (部分数据)：

表 5-2 支撑点数量对范围查询性能的影响 (半径 =0.10)

支撑点数	查询时间 (ms)	距离计算	剪枝数	剪枝率 (%)	结果数
5	0.254	10.6	4994.3	99.89	1.0
10	0.117	10.9	4999.0	99.98	1.0
15	0.041	15.9	4999.0	99.98	1.0
20	0.039	20.9	4999.0	99.98	1.0
25	0.044	25.9	4999.0	99.98	1.0
30	0.042	30.9	4999.0	99.98	1.0

关键发现：

1. **剪枝率随支撑点数量增加而提高：**从 5 个支撑点的 99.89% 提升到 10 个以上的 99.98%。更多的支撑点提供了更多的剪枝机会。
2. **查询时间先降后稳：**从 5 个支撑点的 0.254ms 降至 20 个支撑点的 0.039ms，之后趋于稳定。这是因为剪枝效果提升的同时，剪枝判断开销也在增加。
3. **最优支撑点数量：**对于 5000 个数据点的数据集，15-20 个支撑点达到最佳性能平衡点。
4. **查询半径的影响：**较小的半径 (如 0.05) 获得更高的剪枝率 (接近 100%)，因为满足查询条件的对象更少。

5.3.2 kNN 查询性能

实验结果 (部分数据)：

表 5-3 支撑点数量对 kNN 查询性能的影响 (k=10)

支撑点数	查询时间 (ms)	距离计算	剪枝数	剪枝率 (%)
5	0.422	4506.1	498.9	9.98
10	0.482	4496.5	513.5	10.27
15	0.317	4470.8	544.2	10.88
20	0.215	4463.6	556.4	11.13
25	0.257	4456.9	568.1	11.36
30	0.206	4454.5	575.5	11.51

关键发现：

1. **kNN 查询的剪枝率远低于范围查询：**约 10%-12%，远低于范围查询的 99% 以上。这是因为 kNN 查询需要精确找到 k 个最近邻，动态半径初期较大，剪枝效果有限。

2. **支撑点数量增加带来的提升有限**：从 5 个到 30 个支撑点，剪枝率仅提升 1.5 个百分点，查询时间下降约 50%。

3. **k 值的影响**：较小的 k 值 (如 k=5) 获得更高的剪枝率 (约 12%)，因为动态半径缩小得更快。

5.4 支撑点选择策略对性能的影响分析

本实验对比 RANDOM 和 FFT 两种支撑点选择策略的性能差异。

实验配置：

- 数据集大小：5000 个向量
- 支撑点数量：20
- 选择策略：RANDOM, FFT
- 查询类型：范围查询 (radius=0.1) 和 kNN 查询 (k=10)

范围查询性能对比：

表 5-4 支撑点选择策略对范围查询性能的影响

策略	构建时间 (ms)	查询时间 (ms)	剪枝率 (%)	距离计算
RANDOM	5.877	0.043	99.98	20.9
FFT	54.712	0.044	99.98	20.9

kNN 查询性能对比：

表 5-5 支撑点选择策略对 kNN 查询性能的影响

策略	构建时间 (ms)	查询时间 (ms)	剪枝率 (%)	距离计算
RANDOM	4.150	0.666	10.38	4501.1
FFT	60.750	0.686	11.13	4463.6

关键发现：

1. **FFT 构建时间远高于 RANDOM**：FFT 需要 54.712ms，RANDOM 仅需 5.877ms(范围查询)。FFT 的 $O(nk^2)$ 复杂度导致构建开销大。
2. **查询性能差异不明显**：在本数据集上，两种策略的查询时间和剪枝率差异很小。这可能是因为 20 维均匀分布数据较为规则，随机选择的支撑点也能获得较好的覆盖。
3. **策略选择建议**：
   - 对于一次性查询：RANDOM 策略更合适，构建快速
   - 对于多次查询：FFT 策略的构建开销可以分摊，且在聚类数据上效果更稳定

5.5 查询性能对比：线性扫描 vs. Pivot Table 索引

本实验对比线性扫描和 Pivot Table 索引在不同数据集规模下的性能表现。

5.5.1 范围查询性能对比

实验配置：

- 数据集大小：1000, 5000, 10000
- 支撑点数量：20(FFT 策略)
- 查询半径：0.05, 0.10, 0.15, 0.20

实验结果：

表 5-6 范围查询：线性扫描 vs Pivot Table(半径 =0.10)

数据集大小	线性扫描 (ms)	索引查询 (ms)	加速比	剪枝率 (%)
1000	0.132	0.010	13.83x	99.90
5000	0.313	0.056	5.57x	99.98
10000	1.480	0.151	9.82x	99.99

关键发现：

1. **Pivot Table 显著加速范围查询**：加速比在 5.57x-13.83x 之间，数据集越大，加速效果越明显。
2. **剪枝率极高**：范围查询的剪枝率达到 99% 以上，说明三角不等式剪枝非常有效。
3. **查询时间随数据集大小线性增长**：线性扫描的时间复杂度为 $O(n)$，而 Pivot Table 通过剪枝将实际计算降至 $O(k + \alpha n)$，其中 α 很小。

5.5.2 kNN 查询性能对比

实验结果：

表 5-7 kNN 查询：线性扫描 vs Pivot Table(k=10)

数据集大小	线性扫描 (ms)	索引查询 (ms)	加速比	剪枝率 (%)
1000	0.155	0.071	2.18x	11.28
5000	0.313	0.204	1.53x	11.13
10000	2.408	1.738	1.39x	12.77

关键发现：

1. **kNN 查询加速有限**：加速比仅 1.39x-2.18x，远低于范围查询。这是因为 kNN 查询的剪枝率较低 (约 11%-13%)，大部分对象仍需计算实际距离。
2. **动态半径策略的局限**：kNN 查询的动态半径初期较大，剪枝效果不如固定半径的范围查询。
3. **优化空间**：可以考虑预先估计 k 近邻的距离范围，或采用增量式查询策略来提升剪枝效果。

5.6 性能分析总结

基于以上实验，我们得出以下结论：

1. **支撑点数量**：15-20 个支撑点是 5000-10000 规模数据集的最佳平衡点。过少剪枝不足，过多判断开销增大。

2. **支撑点选择策略**:FFT 策略在聚类数据上更稳定,但均匀分布数据上与 RANDOM 差异不大。应根据数据特征和查询频率选择。

3. **范围查询优化**：Pivot Table 对范围查询的加速效果显著 (5-13 倍)，剪枝率达 99% 以上，是理想的索引方法。

4. **kNN 查询优化**：Pivot Table 对 kNN 查询的加速有限 (1.4-2 倍)，剪枝率仅约 11%，需要更先进的索引结构 (如 VP-tree, M-tree) 来进一步优化。

5. **实际应用建议**：

   - 对于范围查询为主的应用，强烈推荐使用 Pivot Table 索引
   - 对于 kNN 查询为主的应用，可以考虑结合多种索引策略
   - 数据规模越大，索引带来的性能提升越明显

6. 总结与展望

6.1 工作总结

本实验报告在 Assignment 1 构建的度量空间数据管理基础框架之上，成功实现了相似性查询与索引功能，取得了以下成果：

6.1.1 核心功能实现

1. **线性扫描查询**：

   - 实现了范围查询 (Range Query)、k 近邻查询 (kNN Query) 和多样化 k 近邻查询 (dkNN Query) 三种基本查询算法
   - 算法设计通用，支持所有满足度量空间性质的数据类型
   - 作为索引查询的正确性验证基准

2. **Pivot Table 索引**：

   - 实现了基于三角不等式的剪枝优化索引结构

- 支持 RANDOM、FFT、CENTER、BORDER 四种支撑点选择策略
- 实现了基于索引的范围查询和 kNN 查询算法
- 在范围查询上取得了显著的性能提升 (5-13 倍加速)

3. **系统架构：**

- 保持了 Assignment 1 的分层设计，查询和索引模块与核心抽象无缝集成
- 遵循” 依赖倒置” 和” 开闭原则”，易于扩展新的查询类型和索引结构
- 代码结构清晰，注释完善，便于理解和维护

6.1.2 正确性验证

- 设计了易于验证的小规模测试用例，输出详细的计算过程
- 验证了 Pivot Table 索引查询结果与线性扫描完全一致
- 通过 JUnit 测试框架实现了自动化测试，确保系统稳定性
- 验证了度量空间三大性质 (非负性、对称性、三角不等性) 在实现中的正确应用

6.1.3 性能分析成果

- 系统分析了支撑点数量对查询性能的影响，发现 15-20 个支撑点是最佳平衡点
- 对比了 RANDOM 和 FFT 两种支撑点选择策略，揭示了它们的适用场景
- 量化了 Pivot Table 索引的加速效果：范围查询 5-13 倍，kNN 查询 1.4-2 倍
- 揭示了范围查询和 kNN 查询在剪枝效果上的显著差异 (99% vs 11%)

6.1.4 理论理解深化

通过本次实验，我们深入理解了以下核心概念：

- **度量空间通用性**：相同的算法可以处理向量、蛋白质序列等多种数据类型，只需定义满足度量空间性质的距离函数

- **三角不等式的威力**：看似简单的数学性质，在索引优化中发挥了关键作用，将距离计算从 $O(n)$ 降至 $O(k + \alpha n)$
- **剪枝策略的局限性**：Pivot Table 对范围查询效果显著，但对 kNN 查询效果有限，不同查询类型需要不同的优化策略
- **通用与专用的权衡**：通用的度量空间索引牺牲了部分性能，但换来了代码复用和系统的可扩展性

6.2 系统不足与改进方向

尽管本系统实现了预期目标，但仍存在一些不足之处，可在未来工作中改进：

6.2.1 性能优化

1. **kNN 查询优化**：
   - 当前 Pivot Table 对 kNN 查询的加速效果有限 (仅 1.4-2 倍)
   - 可以引入更先进的索引结构，如 VP-tree(Vantage Point Tree)、M-tree 或 MVP-tree
   - 可以采用预估 k 近邻距离范围的策略，缩小初始动态半径
2. **支撑点选择优化**：
   - 当前 FFT 策略的构建开销较大 ($O(nk^2)$)
   - 可以采用抽样式 FFT，在样本上选择支撑点，降低复杂度
   - 可以研究自适应支撑点选择算法，根据数据分布动态调整
3. **并行化**：
   - 当前实现是单线程的，未充分利用多核处理器
   - Pivot Table 的距离表构建天然支持并行化
   - 范围查询的线性扫描部分也可以并行化

6.2.2 功能扩展

1. **更多查询类型：**
    - 实现反向 k 近邻查询 (Reverse kNN)
    - 实现范围内的多样化查询
    - 实现 Top-k 聚合查询
2. **更多索引结构：**
    - 实现 VP-tree、M-tree 等树状索引结构
    - 实现 LAESA(Linear Approximating and Eliminating Search Algorithm)
    - 对比不同索引结构在各种数据和查询场景下的性能
3. **动态数据支持：**
    - 当前索引是静态的，不支持数据的增删改
    - 可以设计增量式索引维护算法
    - 研究批量更新策略，平衡更新开销和查询性能

6.2.3 实验深化

1. **更多数据集：**
    - 当前仅测试了均匀分布的向量数据
    - 应测试聚类数据、真实地理数据 (Texas, Hawaii) 和蛋白质序列数据
    - 分析不同数据分布对索引性能的影响
2. **更大规模：**
    - 当前最大测试规模为 10000 个数据点
    - 应测试 100K、1M 规模的数据集
    - 分析索引的可扩展性

3. **理论分析：**

   - 当前主要是实验性能分析
   - 可以进行理论复杂度分析和期望性能推导
   - 研究剪枝率的上界和下界

6.2.4 工程实践

1. **性能优化：**

   - 采用更高效的数据结构 (如 Java NIO 的 ByteBuffer)
   - 优化距离计算的实现 (如 SIMD 向量化)
   - 引入缓存机制，避免重复计算

2. **可视化：**

   - 开发可视化工具，直观展示剪枝过程
   - 绘制性能曲线图，辅助参数调优
   - 可视化支撑点分布和查询过程

3. **易用性：**

   - 提供命令行工具和图形界面
   - 编写详细的 API 文档和使用示例
   - 支持配置文件，简化系统配置

6.3 展望

度量空间数据管理作为一种通用的数据处理范式，在大数据时代具有重要的理论价值和应用前景。通过本次实验，我们不仅实现了基本的相似性查询和索引功能，更重要的是理解了” 求同存异” 的通用性思想和三角不等式的数学之美。

未来，随着人工智能和大数据应用的发展，度量空间技术有望在图像检索、推荐系统、生物信息学等领域发挥更大作用。我们将继续深入研究度量空间索引的理论和实践，为构建高效、通用的大数据管理系统贡献力量。文章涉及的所有代码均已上传至 GitHub 仓库：https://github.com/sylvanding/BigDataGenhierarchy_Jixiang_20251116。